

Лабораторная работа № 5 по курсу дискретного анализа: Динамическое программирование

Выполнил студент группы М8О-3086-22 МАИ *Караев Тариел Жоомартбекович*.

Условие

Реализовать поиск подстрок в тексте с использованием суффиксного дерева. Суффиксное дерево можно построить за $O(n^2)$ наивным методом.

Метод решения

Наивный способ представляет собой compact trie, который построен по всем суффиксам текущей строки, сначала помещается несобственный суффикс $S[1..m]$, затем последовательно вводятся суффиксы $S[i..m]$ для i от 2 до m . При этом мы должны сохранить свойства суффиксного дерева:

- Trie должен оставаться compact trie
- В наивном алгоритме при вставке очередного суффикса сравниваем первую букву

Количество листьев в trie совпадает с количеством суффиксов в строке суффикса и детей корня, если совпадений нет, то есть из корня мы не можем перейти ни по одному из ребер, то вставляем весь суффикс от корня. Иначе, идем по совпавшему ребру до того момента, пока встретим несовпадение. Тогда нам будет необходимо сделать split текущего ребра, то есть уменьшить его правую границу, создать 2 новых ребра, одно из которых будет отвечать за продолжение исходной дуги, ему необходимо будет перекопировать детей, а второе будет являться оставшейся частью суффикса. Если мы дошли до конца ребра и не встретили несовпадений, то ищем среди детей этого ребра букву, по которой можем перейти дальше, если общая буква существует, то повторяем алгоритм заново, если нет, то вставляем оставшийся суффикс от текущего ребра. Поиск паттерна работает аналогично вставке, только когда мы встречаем несовпадение, то просто останавливаем цикл, если же мы не встретили несовпадений и при этом паттерн закончился, то мы нашли вхождение. Чтобы узнать на каких позициях есть вхождение, необходимо от текущей вершины обойти всех детей поиском в глубину. Так мы получим массив, в котором будут храниться индексы вхождений паттерна в текст.

Описание программы

Был реализован Compact trie

Разделение по файлам, описание основных типов данных и функций.
suffix_tree.h,

```

#include <memory>
#include <map>
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>

class SuffixTrie {
private:
    struct Node {
        size_t l;
        size_t r;
        size_t enter;
        std::map<char, std::shared_ptr<Node>> child;
    };

    std::shared_ptr<Node> root;
    std::string text;

    void Insert(size_t l, size_t r);
    static std::shared_ptr<Node> NewNode(size_t l, size_t r, size_t enter);
    void DFS(std::vector<size_t>& res, std::shared_ptr<Node> current);

public:
    explicit SuffixTrie(const std::string& inputText);
    std::vector<size_t> Search(const std::string& pattern);
};

void SuffixTrie::Insert(size_t l, size_t r) {
    auto current = root;
    size_t oldL = l;

    while (l <= r) {
        if (current == root) {
            if (current->child.find(text[l]) != current->child.end()) {
                current = current->child[text[l]];
            } else {
                current->child[text[l]] = NewNode(l, r, oldL);
                break;
            }
        }
    }
}

```

```

    }

    size_t start = current->l;
    size_t finish = current->r;
    bool cut = false;

    for (size_t i = start; (i <= finish) && (l + i - start <= r); ++i) {
        if (text[i] != text[l + i - start]) {
            current->r = i - 1;
            size_t oldEnter = current->enter;

            auto splitNode = NewNode(i, finish, oldEnter);
            splitNode->child = move(current->child);
            current->child.clear();
            current->child[text[i]] = splitNode;

            current->child[text[l + i - start]] = NewNode(l + i - start,
                cut = true;
            break;
        }
    }

    if (!cut) {
        size_t newL = l + finish - start + 1;
        if (current->child.find(text[newL]) != current->child.end()) {
            current = current->child[text[newL]];
        } else {
            current->child[text[newL]] = NewNode(newL, r, oldL);
            break;
        }
        l = newL;
    } else {
        break;
    }
}
}

```

```

std::shared_ptr<SuffixTrie::Node> SuffixTrie::NewNode(size_t l, size_t r, size_t enter) {
    auto vertex = std::make_shared<Node>();
    vertex->l = l;
    vertex->r = r;
}

```

```

    vertex->enter = enter;
    return vertex;
}

void SuffixTrie::DFS(std::vector<size_t> &res, std::shared_ptr<SuffixTrie::Node> current) {
    if (current->child.empty()) {
        res.push_back(current->enter);
    }
    for (const auto &[first, child] : current->child) {
        DFS(res, child);
    }
}

SuffixTrie::SuffixTrie(const std::string &inputText) {
    text = inputText + '$';
    root = std::make_shared<Node>();
    for (size_t i = 0; i < text.length(); ++i)
        Insert(i, text.length() - 1);
}

std::vector<size_t> SuffixTrie::Search(const std::string &pattern) {
    std::vector<size_t> res;
    auto current = root;
    size_t l = 0;
    size_t r = pattern.length() - 1;
    bool flag = false;

    while (l <= r) {
        if (current == root) {
            if (current->child.find(pattern[l]) != current->child.end()) {
                current = current->child[pattern[l]];
            } else {
                break;
            }
        }

        size_t start = current->l;
        size_t finish = current->r;
        for (size_t i = 0; (start + i <= finish) && (i + l <= r); ++i) {
            if (pattern[i + l] != text[start + i]) {
                flag = true;
                break;
            }
        }
    }
}

```

```

    }
}

    if (!flag) {
        l = l + finish - start + 1;
        if (l > r) {
            break;
        }
        if (current->child.find(pattern[l]) != current->child.end()) {
            current = current->child[pattern[l]];
        } else {
            break;
        }
    } else {
        break;
    }
}

if ((l > r) && (!flag) && (!pattern.empty())) {
    DFS(res, current);
}

return res;
}

```

```

int main() {
    std::string text;
    std::cin >> text;
    SuffixTrie st(text);

    std::string pattern;
    size_t counter = 1;
    while (std::cin >> pattern) {
        std::vector<size_t> res = st.Search(pattern);
        if (!res.empty()) {
            printf("%zu: ", counter);
            std::sort(res.begin(), res.end());
            for (size_t i = 0; i < res.size(); ++i) {
                if (i != 0) {
                    printf(", ");
                }
                printf("%zu", res[i] + 1);
            }
        }
    }
}

```

```

        }
        printf("\n");
    }
    ++counter;
}

return 0;
}

```

Функции-члены класса: Приватные:

1. Insert - вставляет часть строки с индекса l по индекс r в trie
2. NewNode - статический метод для создания узла без детей
3. DFS - функция, которая обходит всех детей узла и сохраняет номера листьев переданный вектор

Публичные

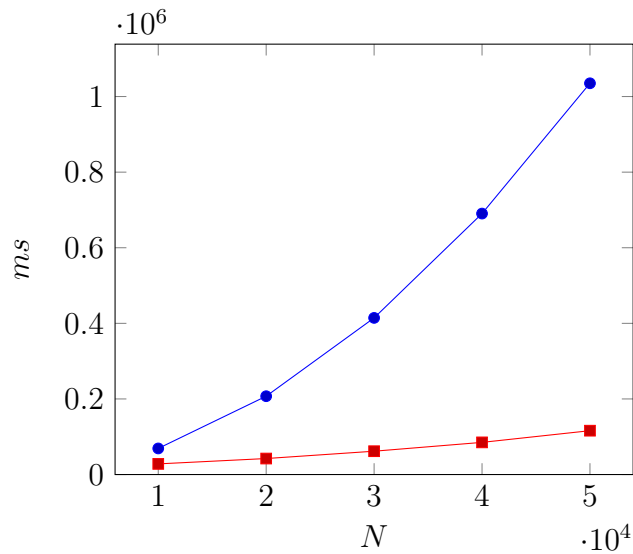
1. SuffixTrie - конструктор, принимающий одну строку. Он добавляет сентинел к строке и вставляет все префиксы строки в trie с помощью Insert
2. Search - возвращает вектор с индексами всех вхождений строки в текст

Дневник отладки

В основном возникали проблемы с реализацией алгоритмов и получением неверных ответов.

Тест производительности

Длина текста будет фиксированной и равной 10000. Попробуем искать все вхождения N строки из 10 символов в этом тексте используя последовательный std::find и используя suffix_trie



Таким образом видим, что суффиксное дерево отлично подходит для поиска большого количества паттернов в одном неизменном тексте. Поиск одного паттерна через `std::find` имеет сложность $O(n \times k + m)$ где k - количество вхождений паттерна, а поиск через суффиксное дерево $O(m + k)$. $O(m)$ на спуск и $O(k)$ на обход поддеревы.

0.1 Выводы

В ходе выполнения лабораторной работы было реализовано суффиксное дерево. В отличие от других алгоритмов поиска подстроки в строке, суффиксное дерево позволяет заранее обработать текст, а затем искать паттерны за линейное время. Стоит отметить, что было реализовано сжатое суффиксное дерево, строящееся наивным методом.